Formalization of Quantifier Elimination Methods in Lean

Ruth Plümer

Geboren am 8. Dezember 2004 in Witten

August 4, 2026

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Philipp Hieronymi

Zweitgutachter: Prof. Dr. Floris van Doorn

Mathematisches Institut

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Acknowledgements

I would like to thank Prof. Floris van Doorn for his helpful advice during the Lean Hacking Sessions, in particular his help with avoiding issues with type coercions in the implementation of partial isomorphisms.
Moreover, I appreciated Aaron Liu and Matteo Cipollina taking the time to answer my questions on the Lean Zulip forum when I was struggling to understand the mathlib definition of bounded formulas.
Additionally, I would like to express my thanks to my friends Max Keßler and Florian Jäniche for their incredible patience in listening to my countless Lean problems over and over again and the advice they offered when my questions did not end with "nevermind, I just figured it out".
Last, but not least, this project would not have been possible without the advice and encouragement from my advisor, Prof. Philipp Hieronymi.

Abstract

This Bachelor's thesis aims to present a formalization of the model-theoretic concept of quantifier elimination as well as a well-known quantifier elimination criterion via the proof assistant Lean, built on top of on mathlib, Lean's mathematical library. By doing so, we provide a basis for the formalization of other quantifier elimination criteria and other more advanced model theoretic results in Lean.

Furthermore, we discuss the Separating Types Theorem, a result about model-theoretic types, in Lean and present a formalization, using a technical consequence of the Compactness Theorem.

While only selected parts of the Lean code are presented in this thesis, the complete formalization project can be accessed under `https://github.com/RuthP628/QuantifierElimination`.

Deutschsprachige Zusammenfassung

Diese Bachelorarbeit verfolgt das Ziel, eine Formalisierung des modelltheoretischen Konzeptes der Quantorenelimination in Lean zu präsentieren, die auf Leans mathematischer Bibliothek mathlib aufbaut. Diese bietet eine Grundlage für mögliche Formalisierungen anderer Quantoreneliminationskriterien sowie weiterführender modelltheoretischer Resultate.

Weiterhin diskutieren wir das Separating Types Theorem, ein Resultat über modelltheoretische Typen, und formalisieren einen Beweis dieses Satzes, der ein technisches Korollar aus dem Kompaktheitssatz verwendet.

Während nur ausgewählte Teile des Lean-Codes in dieser Arbeit präsentiert werden, ist das vollständige Formalisierungsprojekt unter `https://github.com/RuthP628/QuantifierElimination` abrufbar.

Contents

1 Introduction

1.1 Mathematical motivation

As W. Hodges explains in [Hod97], model theory is "about the classification of mathematical structures, maps and sets by the means of logical formulas". This might already suggest reducing model-theoretic questions to the case of quantifier-free formulas whenever possible, since these are certainly the logical formulas that are the easiest to understand and classify.

Indeed, the method of quantifier elimination is a powerful tool in model theory achieving exactly that. The following definition is taken from [TZ12]:

Definition 1.1 (Quantifier elimination)**.** Let $\mathcal{L}$ be a language and T be an $\mathcal{L}$-theory. T is said to have *quantifier elimination* (short: "*T has QE*") if for every $\mathcal{L}$-formula $\varphi(x_1, \ldots, x_n)$, there is a quantifier-free $\mathcal{L}$-formula $\psi(x_1, \ldots, x_n)$ such that φ and ψ are equivalent modulo T.

In the past decades, a number of theories have been proven to admit quantifier elimination, including but not limited to the theory of $(\mathbb{N}, +, 0, 1, (\equiv_k)_{k>0})$, often called *Presburger arithmetic* [Sta84], the theory of dense linear orders [Bra+07], the theory of real closed fields [Bra+07], the theory of algebraically closed fields [Pas16], the theory of abelian groups [Szm55], and the theory of the Rado graph [Fuc10].

The quantifier elimination property has been used as a means to prove different interesting model-theoretic properties for different theories, including completeness (as first demonstrated by Mojżesz Presburger for Presburger arithmetic [Sta84]) and decidability (as demonstrated by Büchi for Presburger arithmetic [Ric66]). It should be noted that QE alone is neither necessary not sufficient for either of these properties, but the restriction to the case of quantifier-free formulas often trivializes the questions of completeness and decidability.
Moreover, quantifier elimination has been used as a tool to classify non-forking extensions of theories, as done by Martin-Pizarro for $DCF_{0,m}$ [Mar24].

Hence, it is not surprising that model theorists have developed many quantifier elimination criteria. In this thesis, we are going to focus on a particular QE criterion, namely the following one, which was taken from [Hie26]:

Theorem 1.2. *Let $\mathcal{L}$ be a language and T be an $\mathcal{L}$-theory such that for any $\mathcal{L}$-structures $\mathcal{M}, \mathcal{N} \models T$, the set of partial isomorphisms between $\mathcal{M}$ and $\mathcal{N}$ with finite domain is a back-and-forth-system.*
Then, T has quantifier elimination.

The main goal of this thesis is to formalize the proof of the theorem above using the interactive theorem prover Lean. In the course of this, we formalize the notion of quantifier elimination as well as the definitions of partial isomorphisms and back-and-forth-systems. Moreover, we prove the *Separating Types Theorem*, a consequence of the compactness theorem necessary to prove Theorem 1.2, in Lean.

The mathematical content of this thesis is mostly standard material covered in an introductory model theory course. If not indicated otherwise, the (non-formal) definitions, theorems and proofs in this thesis will be based on [Hie26].

1.2 Lean and Mathlib

The interactive theorem prover and functional programming language Lean was originally developed by Leonardo de Moura at Microsoft Research in 2013 [MU21]. The formalization part of this thesis is written in lean4, its most recent version as of writing. The Lean software is open source and accessible via GitHub [Com].

Mathematically, when using the Lean theorem prover, results are implemented using *dependent type theory*. Basically, this means that all objects belong to some *type*. These types can be thought of as data types in other programming languages like C++; for example, the object `1` might have type `Nat`, `Int` or `Real`, depending on the context. Types themselves also have types, living in a "higher universe", which leads to an infinite *universe hierarchy*.

Additionally, there is a number of standard ways on how to construct new types from existing ones. Most notably, for types `A` and `B`, there is an *arrow type* $\texttt{A} \to \texttt{B}$. Inhabitants of $\texttt{A} \to \texttt{B}$ can be thought of as "ways to construct an inhabitant of type `B` if given an inhabitant of `A`", so, roughly speaking, functions. (Note that `A` and `B` do not need to be sets or "set-like structures"!). Similarly, if given types `A` and `B`, one can construct a type $\texttt{A} \times \texttt{B}$ which basically includes "pairs of inhabitants" of a and b.

In Lean, mathematical statements are regarded as inhabitants of the *proposition type* `Prop`. Moreover, every mathematical statement *is a type in itself* and inhabitants of it are proofs of the statement, i.e. proving a statement A in Lean means constructing an element of type A. Since we are not really interested in the differences between certain proofs of the same statement A, but only in the truth value of A, all proofs of A are regarded as equal in Lean (*proof irrelevance*).

Now, one can see that the statement "given a proof of `A`, we can construct

a proof of `B`" corresponds to a proof of the logical implication from `A` to `B`. Hence, the arrow type `A → B` can be thought of as a logical implication if `A` and `B` are of type `Prop`. Similarly, in this case, the product type `A × B` can be thought of as the logical conjunction of `A` and `B`.
This striking correspondence comes from the *Curry-Howard isomorphism*, which has first been proposed by Haskell Curry in 1934 [Cur34] and can be thought of as "a correspondence between proofs and computer programs".

In theory, all mathematical proofs can be expressed purely as type-theoretic terms. However, in practice, writing such proofs in Lean would quickly become extremely tedious and complicated, moreover, purely functional proofs are very hard to read to most humans. For this reason, Lean also supports a *tactic mode*, which allows users to enter proofs in a more intuitive form using *tactics*, like the `have`-tactic for introducing a new local hypothesis.

In particular, Lean offers several *automation tactics* that allow users to skip obvious steps in a proof, like the `omega`-tactic that automatically decides statements expressable in Presburger arithmetic (recall: this is decidable!).

A detailed explanation of dependent type theory and Lean is beyond the scope of this thesis. For a comprehensive introduction, see [Avi+21]. For a more hands-on introduction to formalization with Lean, see [AM20].

As most Lean formalization project, this thesis is heavily based on Lean's mathematical library *mathlib*. Mathlib is a large community-driven open-source library of definitions, theorems and mathematical proofs that aims to cover foundational results in mathematics. The code of mathlib is developed and can be accessed on the software development platform GitHub [Com17].

The scope of mathlib is often explained to be "the contents of a typical undergraduate or graduate-level mathematics course" [Com23]. Nevertheless, Mathlib has been used as a foundation for a number of large-scale formalization projects of current research questions, the code of many of which has gradually been adopted to mathlib afterwards. The most notable of these projects is probably the *Liquid Tensor Experiment*, a formalization of a key result in condensed mathematics by Peter Scholze and Dustin Clausen, which was lead by Johan Commelin and Adam Topaz [Com22].

Originally created in 2017 by Mario Carneiro and Johannes Hölzl for lean3 [Com20c], the most recent Lean version at the time, mathlib has heavily grown over time. At the time of writing, it consists of 134691 definitions and 283110 (proven) theorems, written by 772 different contributors [Com20b].

Ultimately, the aim of this thesis is also to contribute code to Mathlib in order to help with making it more comprehensive in the area of basic model theory. However, as of August 2026, most of the code written in the course of this thesis does not yet adhere to mathlib's relatively strict naming conventions and style guide. Therefore, at the time of writing, the code is only accessible via a standalone GitHub repository depending on mathlib under `https://github.com/RuthP628/QuantifierElimination`.

1.3 Previous Works

Due to the nature of mathlib as a community-driven effort, some mathematical areas receive more attention than others. Indeed, model theory proves to be a somewhat underdeveloped area in mathlib, with only 34 out of the – at the time of writing – 9000 files of mathlib belonging to it [Com20a] (for a comparison, 1089 out of the 9000 files belong to category theory alone, not including algebraic geometry, algebraic topology and condensed mathematics). Beyond that, only one of the 28 mathlib maintainers, namely Floris van Doorn, lists model theory as one of his interests [Com17].

Nevertheless, a number of standard model theory results have been formalized in Lean, including but not limited to the compactness theorem [And21], the Löwenheim-Skolem theorems [And22c], the Taski-Vaught test [And22a] and the Łos-Vaught test [And21]. Of particular importance to this thesis is the *Flypitch project*, a formalization of the independence of the continuum hypothesis by Han and van Doorn [HD19] [HD20]. In fact, at the time of writing, most of the model theory code in mathlib is adapted from the Flypitch project, which was originally written in lean3. Moreover, we will use a number of results developed by Anderson and Kolly for the formalization of Fraïssé limits [Kol25]. Another project that should be mentioned here is the *Formalized Formal Logic Repository*, an implementation of Gödel's completeness and incompleteness theorems as well as a number of results from intuitionistic, modal and interpretability logic in Lean that is also based on mathlib [NS24].

2 Overview of Model Theory in Mathlib

Now, we will present an overview of the implementations of basic model theoretic results in mathlib that are used in the author's original formalization work. The mathlib code containing the implementations of languages, structures, terms, formulas, models and realizations is adapted from Han's and van Doorn's work in the Flypitch project [HD19], while the formalization of model-theoretic types was developed by Anderson [And22d].

2.1 Languages and Structures

In mathlib, a `Language` is implemented as a structure with two attributes: a function `Functions`, assigning every natural number a type of functions of this arity, and a function `Relations`, assigning every natural number a type of relations of this arity.

```
-- Code in the file ModelTheory/Basic, written by Han and van Doorn
structure Language where
  /-- For every arity, a `Type u` of functions of that arity -/
  Functions : ℕ → Type u
  /-- For every arity, a `Type v` of relations of that arity -/
  Relations : ℕ → Type v
```

From now on, $\mathcal{L}$ (resp. `L`) will denote a language.

Note that we can extend a language $\mathcal{L}$ to a new language $\mathcal{L}(\alpha)$ that includes the inhabitants of a type `α` as constant symbols (= functions with arity 0). In Lean, we denote $\mathcal{L}(\alpha)$ by `L[[α]]`:

```
-- Code in the file ModelTheory/LanguageMap
-- Written by Anderson, Han and van Doorn
def withConstants (α : Type w') : Language.{max u w', v} :=
  L.sum (constantsOn α)

@[inherit_doc FirstOrder.Language.withConstants]
scoped[FirstOrder] notation:max L "[[" α "]]" =>
  Language.withConstants L α
```

We can map the symbols of a language $\mathcal{L}$ to the symbols of a language $\mathcal{L}'$ by a language homomorphism:

```
-- Code in the file ModelTheory/LanguageMap
-- Written by Anderson, Han and van Doorn
structure LHom where
  /-- The mapping of functions -/
  onFunction : ∀ ⦃n⦄, L.Functions n → L'.Functions n := by
    exact fun {n} => isEmptyElim
  /-- The mapping of relations -/
  onRelation : ∀ ⦃n⦄, L.Relations n → L'.Relations n := by
    exact fun {n} => isEmptyElim
```

```
@[inherit_doc FirstOrder.Language.LHom]
infixl:10 " →ᴸ " => LHom
```

The last two lines allow us to write `L →ᴸ L'` for a language homomorphism from $\mathcal{L}$ to $\mathcal{L}'$.

For us, the most important language homomorphism will be `lhomWithConstants`, the map embedding `L` in `L[[α]]`:

```
-- Code in the file ModelTheory/LanguageMap
-- Written by Anderson, Han and van Doorn
def lhomWithConstants : L →ᴸ L[[α]] :=
  LHom.sumInl
```

Moreover, for any type M, we can implement an $\mathcal{L}$-structure on M (denoted as `L.Structure M`) by specifying interpretations for function and relation symbols. Here, `funMap` is a map sending n-ary $\mathcal{L}$-functions (i.e. inhabitants of the type `L.Functions n`) to functions $M^n \to M$. Elements of M^n are represented as functions $\{0, \dots, n-1\} \to M$ (i.e. `Fin n → M`). `RelMap` works similarly.

Note: in Lean, the expression `A → B → C` is interpreted as `A → (B → C)`.

```
-- Code in the file ModelTheory/Basic, written by Han and van Doorn.
class Structure where
  /-- Interpretation of the function symbols -/
  funMap : ∀ {n}, L.Functions n → (Fin n → M) → M := by
    exact fun {n} => isEmptyElim
  /-- Interpretation of the relation symbols -/
  RelMap : ∀ {n}, L.Relations n → (Fin n → M) → Prop := by
    exact fun {n} => isEmptyElim
```

From now on, M and N will always be types equipped with an $\mathcal{L}$-structure. In mathematical proofs, we will write $\mathcal{M}$ (resp. $\mathcal{N}$) to denote M (resp. N) together with the structure on it.

2.2 Terms and Formulas

Terms are defined as inductive types:

```
-- Code in the file ModelTheory/Syntax, written by Han and van Doorn
inductive Term (α : Type u') : Type max u u'
  | var : α → Term α
  | func : ∀ {l : ℕ} (_f : L.Functions l) (_ts : Fin l → Term α),
    Term α
```

In natural language, the definition above corresponds to:

Definition 2.1. Let $\mathcal{L}$ be a language and α be a type. Then,

- For every inhabitant x of α, there is a term corresponding to x, called a *variable*.

- For every function symbol $f \in \mathcal{L}$ with arity l and terms $t_1, \ldots, t_l$, $f(t_1, \ldots, t_l)$ is a term.

Note that the definition above does not require elements of `L.Term α` to contain all variables in `α`. Hence, the following function takes in a term t with variables in `α` and returns the finite set of variables in `L.Term α` that actually occur in t:

```
-- Code in the file ModelTheory/Syntax, written by Han and van Doorn
def varFinset [DecidableEq α] : L.Term α → Finset α
  | var i => {i}
  | func _f ts => univ.biUnion fun i => (ts i).varFinset
```

For this to work, equality of two different elements of the type `α` has to be decidable, meaning that it has to be possible to generate an instance of `DecidableEq α` from the local context.

In Lean, formulas are implemented using the auxiliary definition `BoundedFormula`. For a natural number `n`, a `BoundedFormula α n` is a formula with free variables in the type `α` and `n` variables over which we will quantify later during the procedure of constructing a formula. Moreover, a `Sentence` is defined to be a formula without free variables.

```
-- Code in the file ModelTheory/Syntax, written by Han and van Doorn
inductive BoundedFormula (α : Type u') : ℕ → Type max u v u'
  | falsum {n} : BoundedFormula α n
  | equal {n} (t₁ t₂ : L.Term (α ⊕ (Fin n))) : BoundedFormula α n
  | rel {n l : ℕ} (R : L.Relations l)
    (ts : Fin l → L.Term (α ⊕ (Fin n))) : BoundedFormula α n
  /-- The implication between two bounded formulas. -/
  | imp {n} (f₁ f₂ : BoundedFormula α n) : BoundedFormula α n
  /-- The universal quantifier over bounded formulas. -/
  | all {n} (f : BoundedFormula α (n + 1)) : BoundedFormula α n

abbrev Formula := L.BoundedFormula α 0

abbrev Sentence := L.Formula Empty
```

Again, the definition above does not require all inhabitants of `α` to appear in objects of type `L.BoundedFormula α n`. Hence, the following function takes in elements `φ` of the type `L.BoundedFormula α n` and returns the finite set of free variables actually occuring in `φ`:

```
-- Code in the file ModelTheory/Syntax, written by Han and van Doorn
def freeVarFinset [DecidableEq α] :
  ∀ {n}, L.BoundedFormula α n → Finset α
    | _n, falsum => ∅
    | _n, equal t₁ t₂ => t₁.varFinsetLeft ∪ t₂.varFinsetLeft
    | _n, rel _R ts => univ.biUnion fun i => (ts i).varFinsetLeft
    | _n, imp f₁ f₂ => f₁.freeVarFinset ∪ f₂.freeVarFinset
    | _n, all f => f.freeVarFinset
```

Again, we can only define this map if equality in `α` is decidable.

A construction we are going to use several times in the formalization project is `BoundedFormula.iSup` (resp. `Formula.iSup`), which is the "disjunction of finitely many formulas":

```
-- Code in ModelTheory/Syntax, written by Anderson, Han and van Doorn
noncomputable def iSup [Finite β] (f : β → L.BoundedFormula α n) :
  L.BoundedFormula α n :=
    let _ := Fintype.ofFinite β
    ((Finset.univ : Finset β).toList.map f).foldr (· ⊔ ·) ⊥
```

In this definition, `f` is a function indexing $\mathcal{L}$-formulas by a finite type `β`. To construct the disjunction, one lists the inhabitants of `β` in an arbitrary order and then recursively takes the disjunction of their images under `f`, starting from the right. Here, the empty disjunction is defined to be `⊥`. The "conjunction of finitely many formulas", `BoundedFormula.iInf` (resp. `Formula.iInf`) is defined analogously, except that the empty conjunction is defined to be `⊤`.

Finally, note that formulas with free variables in α can also be interpreted as $\mathcal{L}(\alpha)$-sentences. In Lean, this is achieved by the bijection `equivSentence`:

```
-- Code in the file ModelTheory/Syntax, written by Han and van Doorn
def equivSentence : L.Formula α ≃ L[[α]].Sentence :=
  (BoundedFormula.constantsVarsEquiv.trans
  (BoundedFormula.relabelEquiv (Equiv.sumEmpty _ _))).symm
```

2.3 Models and Realizations of Formulas

In an $\mathcal{L}$-structure $\mathcal{M}$, we can evaluate terms by assigning a value in M to every variable in α and evaluating function symbols by functions on M of the same arity:

```
-- Code in the file ModelTheory/Semantics
-- Written by Andersen, Han and van Doorn
def Term.realize (v : α → M) : ∀ _t : L.Term α, M
  | var k => v k
  | func f ts => funMap f fun i => (ts i).realize v
```

Similarly, we can inductively define an evaluation of bounded formulas (and therefore also formulas and sentences) by assigning a value in M to every free variable in α and bound variable in `Fin n`. Here, l-ary relation symbols get mapped to l-ary relations on M.

If an `L.Sentence` `φ` evaluates to `True` in an `L.Structure M`, we denote this by $\mathcal{M} \models \varphi$ or `M ⊨ φ`. Moreover, in this thesis, we will write $\mathcal{M} \models \varphi(a)$ if $a : \alpha \to M$ realizes φ.

```
-- Code in the file ModelTheory/Semantics
-- Written by Anderson, Han and van Doorn
def Realize : ∀ {l} (_f : L.BoundedFormula α l) (_v : α → M)
 (_xs : Fin l → M), Prop
  | _, falsum, _v, _xs => False
  | _, equal t₁ t₂, v, xs =>
     t₁.realize (Sum.elim v xs) = t₂.realize (Sum.elim v xs)
  | _, rel R ts, v, xs => RelMap R fun i =>
    (ts i).realize (Sum.elim v xs)
  | _, imp f₁ f₂, v, xs => Realize f₁ v xs → Realize f₂ v xs
  | _, all f, v, xs => ∀ x : M, Realize f v (snoc xs x)

nonrec def Realize (φ : L.Formula α) (v : α → M) : Prop :=
  φ.Realize v default

nonrec def Sentence.Realize (φ : L.Sentence) : Prop :=
  φ.Realize (default : _ → M)

@[inherit_doc Sentence.Realize]
infixl:51 " ⊨ " => Sentence.Realize
```

The API of these definitions contains lemmas stating that this evaluation is well-behaved, in particular, that it commutes with negation, conjunction and disjunction of formulas as well as language homomorphisms.

Finally, a *theory* is a set of $\mathcal{L}$-sentences:

```
-- Code in the file ModelTheory/Syntax, written by Han and van Doorn
abbrev Theory := Set L.Sentence
```

We define a *model* of a theory T to be an $\mathcal{L}$-structure in which all sentences in T evaluate to `True` :

```
-- Code in the file ModelTheory/Semantics
-- Written by Anderson, Han and van Doorn
class Theory.Model (T : L.Theory) : Prop where
  realize_of_mem : ∀ φ ∈ T, M ⊨ φ

@[inherit_doc Theory.Model]
infixl:51 " ⊨ " => Theory.Model
```

The last two lines allow us to write `T ⊨ M` instead of `T.Model M`.
In this thesis, we will use the notation $T \models \varphi$ to express that the $\mathcal{L}$-sentence φ is realized in every model of T.

Furthermore, we define `T.ModelType` to be the type of nonempty models of `T`:

```
-- Code in the file ModelTheory/Bundled, written by Anderson
structure ModelType where
  /-- The underlying type for the models -/
  Carrier : Type w
  [struc : L.Structure Carrier]
  [is_model : T.Model Carrier]
  [nonempty' : Nonempty Carrier]
```

Somewhat ambiguously, in the literature, the notation $T \models \varphi$ is also used for $\mathcal{L}$-formulas with free variables x_1, ..., x_n as an abbreviation for $T \models \forall x_1 \dots \forall x_n \varphi$. In Lean, the notation for this is `T ⊨[b] φ`:

```
-- Code in the file ModelTheory/Satisfiability, written by Anderson
def ModelsBoundedFormula (φ : L.BoundedFormula α n) : Prop :=
  ∀ (M : ModelType.{u, v, max u v w} T) (v : α → M) (xs : Fin n → M),
    φ.Realize v xs

@[inherit_doc FirstOrder.Language.Theory.ModelsBoundedFormula]
infixl:51 " ⊨[b] " => ModelsBoundedFormula
```

Note that in Lean, `T ⊨[b] φ` means that for all models `M ⊨ T` and all interpretations `v : α → M`, `xs : Fin n → M` of the variables of `φ`, `φ` is realized by `v` and `xs`. One can easily see that this is equivalent to the standard definition of $T \models \varphi$, as described above.

Finally, the following theorem tells us how `equivSentence` interacts with realization of formulas:

```
-- Code in the file ModelTheory/Semantics
-- Written by Anderson, Han and van Doorn
theorem realize_equivSentence [L.Structure M] [L[[α]].Structure M]
  [(L.lhomWithConstants α).IsExpansionOn M] (φ : L.Formula α) :
    (equivSentence φ).Realize M
    ↔ φ.Realize fun a => (L.con a : M) := by
      rw [← realize_equivSentence_symm_con M (equivSentence φ),
      _root_.Equiv.symm_apply_apply]
```

In the code above, the instances `L.Structure M`, `L[[α]].Structure M` and `(L.lhomWithConstants α).IsExpansionOn M` ensure that there are both an $\mathcal{L}$-structure and an $\mathcal{L}(\alpha)$-structure on $\mathcal{M}$ that agree on the symbols of $\mathcal{L}$. If this is the case, then `M` models `equivSentence φ` if and only if the interpretation of the elements of `α` in `M` realizes `φ`.

2.4 Model-theoretic types

The following definition is based on section 4.2 in [TZ12].

Definition 2.2. Let T be an $\mathcal{L}$-theory. A *(complete) n-type over T* is a set $p(x_1, \ldots, x_n)$ of $\mathcal{L}$-formulas such that $p \cup T$ is satisfiable and for all $\mathcal{L}$-formulas $\varphi(x_1, \ldots, x_n)$, we have either $\varphi \in p$ or $\neg\varphi \in T$.
Moreover, if $\mathcal{M} \models T$ and $b \in M^n$, we define the *type of b* (short: $\mathrm{tp}^{\mathcal{M}}(b)$) to be the set of $\mathcal{L}$-formulas φ s.t. $\mathcal{M} \models \varphi(a)$.

In mathlib, inhabitants of `T.CompleteType α` are defined as maximal $\mathcal{L}(\alpha)$-theories that contain the image of T under `L.lhomWithConstants α`. Here, a maximal theory is a satisfiable theory that contains every $\mathcal{L}$-sentence φ or its negation.

One can easily see that this definition is essentially the definition from above if we choose `α` to be `Fin n`.

```
-- Code in the file ModelTheory/Types, written by Anderson
structure CompleteType where
  /-- The underlying theory -/
  toTheory : L[[α]].Theory
  subset' : (L.lhomWithConstants α).onTheory T ⊆ toTheory
  isMaximal' : toTheory.IsMaximal
```

Consequently, for `v : α → M`, `T.typeOf v` is defined to be the set of $\mathcal{L}(\alpha)$-sentences realized in `M`, if `M` is given an $\mathcal{L}(\alpha)$-structure by assigning every element of `α` its image under `v` as an interpretation.

```
-- Code in the file ModelTheory/Types, written by Anderson
def typeOf (v : α → M) : T.CompleteType α :=
  haveI : (constantsOn α).Structure M := constantsOn.structure v
  { toTheory := L[[α]].completeTheory M
    subset' := model_iff_subset_completeTheory.1
      ((LHom.onTheory_model _ T).2 inferInstance)
    isMaximal' := completeTheory.isMaximal _ _ }
```

Next, we will consider realizations of types.

Definition 2.3. A complete n-type p over T *is realized in* an $\mathcal{L}$-structure $\mathcal{M} \models T$ if and only if there exists $a \in M^n$ s.t. $p = \mathrm{tp}(a)$.

In mathlib, this is formalized as follows:

```
-- Code in the file ModelTheory/Types, written by Anderson
def realizedTypes (T : L.Theory) (M : Type w') [L.Structure M]
  [Nonempty M] [M ⊨ T] (α : Type w) : Set (T.CompleteType α) :=
    Set.range (T.typeOf : (α → M) → T.CompleteType α)
```

Since for all types p, $p \cup T$ is satisfiable, we know that every type p is realized in some model $\mathcal{M} \models T$. In Lean, the formalization of this statement reads:

```
-- Code in the file ModelTheory/Types, written by Anderson
theorem exists_modelType_is_realized_in (T : L.Theory)
  (p : T.CompleteType α) :
    ∃ M : Theory.ModelType.{u, v, max u v w} T, p ∈ T.realizedTypes M α
```

3 Separating Types Theorem

The goal of this section is to formalize the proof of the *Separating Types Theorem*, a basic theorem in model theory that is needed to prove our main result:

Theorem 3.1 (Separating Types Theorem)**.** *Let $\mathcal{L}$ be a language, T an $\mathcal{L}$-theory and $\Sigma(x_1, \ldots, x_n)$ be a set of $\mathcal{L}$-formulas s.t. $\top \in \Sigma$, $\bot \in \Sigma$ and Σ is closed under conjunction and disjunction of formulas. Then, the following are equivalent:*

1. *For every $\mathcal{L}$-formula φ, there is $\psi \in \Sigma$ s.t. $T \models (\varphi \leftrightarrow \psi)$.*
2. *For all distinct n-types p, q over T, there is $\psi \in \Sigma$ s.t. $\psi \in p$ and $\psi \notin q$.*

Remark 3.2. Most applications of the Separating Types Theorem assume Σ to be the set of quantifier-free formulas, which clearly satisfies the assumptions. Then, condition 1 translates to T having quantifier elimination.

3.1 A consequence of the compactness theorem

In order to prove the Separating Types Theorem, we are going to need the following result, which is a consequence of the compactness theorem:

Lemma 3.3. *Let T be an $\mathcal{L}$-theory and $\varphi(x_1, \ldots, x_n)$ be an $\mathcal{L}$-formula such that the following condition is satisfied:*

For all $\mathcal{M} \models T$ and $a \in M^n$ with $\mathcal{M} \models \varphi(a)$, there is $\psi \in \Sigma$ s.t. $\mathcal{M} \models \psi(a)$.

Then, there are finitely many $\mathcal{L}$-formulas $\psi_1, \ldots, \psi_m \in \Sigma$ such that

$$T \models \left(\varphi \rightarrow \bigvee_{i=1}^{m} \psi_i \right)$$

Proof. Let $\mathcal{L}_c$ be the language $\mathcal{L}$ together with constant symbols c_1, ..., c_n. Moreover, consider the $\mathcal{L}_c$-theories $T_\varphi = T \cup \{\varphi(c_1, \ldots, c_n)\}$ and $\Delta := \{\neg\psi(c_1, \ldots, c_n) : \psi \in \Sigma\}$. Note that by assumption, for every model $\mathcal{M} \models T_\varphi$, there is a formula $\psi \in \Sigma$ s.t. $\mathcal{M} \models \psi(c_1, \ldots, c_n)$. Hence, $T_\phi \cup \Delta$ is not satisfiable.
By the compactness theorem, this implies that there is a finite subset $\Delta' \subseteq \Delta$ s.t. $T_\varphi \cup \Delta'$ is not satisfiable.
Hence, there are $\mathcal{L}$-formulas $\chi_1(x_1, \ldots, x_n), \ldots, \chi_m(x_1, \ldots, x_n)$ such that

$$\Delta' = \{\chi_1(c_1, \ldots, c_n), \ldots, \chi_m(c_1, \ldots, c_n)\}$$

Since $T_\varphi \cup \Delta'$ is not satisfiable, it follows that

$$T_\varphi \vDash \neg \left(\bigwedge_{i=1}^{m} \chi_i(c_1, \ldots, c_n) \right).$$

This is equivalent to

$$T_\varphi \vDash \bigvee_{i=1}^{m} \neg\chi_i(c_1, \ldots, c_n).$$

Note that by definition of Σ, there are $\psi_1, \ldots, \psi_m \in \Sigma$ s.t. $\chi_i = \neg\psi_i$ for all $1 \leq i \leq m$. Hence, we know that

$$T \vDash \left(\varphi \to \bigvee_{i=1}^{m} \psi_i \right).$$

□

For the proof in Lean, we are going to work out some steps explicitly that one would omit in a proof in natural language, the first one being:

Lemma 3.4. *Let $\mathcal{L}$ be a language, T be an $\mathcal{L}$-theory and φ, ψ be $\mathcal{L}$-sentences. Then,*

$$(T \cup \{\varphi\}) \models \psi \text{ if and only if } T \models (\varphi \to \psi).$$

In Lean, this translates to:

```
-- Code in the file SeparatingTypes
-- of the author's formalization project
lemma exist_models_iff_models_imp (T : L.Theory) (φ : L.Sentence)
  (ψ : L.Sentence) : (T ∪ {φ}) ⊨ᵇ ψ ↔ T ⊨ᵇ (φ.imp ψ)
```

Additionally, we proved that the bijection `equivSentence` commutes with `iInf` and `iSup`:

```
-- Code in the file SeparatingTypes
-- of the author's formalization project
lemma equivSentence_iInf [Finite α] {β : Type*} (M : Type*)
  [L[[β]].Structure M] (f : α → L.Formula β) :
    M ⊨ equivSentence (iInf f) ↔ ∀ x : α, M ⊨ equivSentence (f x)

lemma equivSentence_iSup [Finite α] {β : Type*} (M : Type*)
  [L[[β]].Structure M] (f : α → L.Formula β) :
    M ⊨ equivSentence (iSup f) ↔ ∃ x : α, M ⊨ equivSentence (f x)
```

Furthermore, we proved that the equivalences $\neg(\varphi \vee \psi) \leftrightarrow (\neg\varphi \wedge \neg\psi)$ and $\neg(\varphi \wedge \psi) \leftrightarrow (\neg\varphi \vee \neg\psi)$ generalize to `iInf` and `iSup`:

```
-- Code in the file SeparatingTypes
-- of the author's formalization project
lemma iInf_iff_not_iSup_not (T : L.Theory) [Finite α] {β : Type*}
  (f : α → L.Formula β) :
    T.Iff (iInf f) (iSup (fun x ↦ (f x).not)).not

lemma iSup_iff_not_iInf_not (T : L.Theory) [Finite α] {β : Type*}
  (f : α → L.Formula β) :
    T.Iff (iSup f) (iInf (fun x ↦ (f x).not)).not
```

The details of these formal proofs are omitted here, but can be found on the author's GitHub.

Now, we are ready to formalize the proof of lemma 3.3. Due to the length of the formal proof, we do not present it on paper, but instead refer to the author's GitHub again.

```
-- Code in the file SeparatingTypes
-- of the author's formalization project
lemma impliesfinitedisj_if {α : Type w} [Finite α]
  (T : L.Theory) (ψ : L.Formula α) (Sigma : Set (L.Formula α))
  (hSigma : ∀ (M : ModelType.{u, v, max (max u v) w} T)
  (a : α → M.Carrier), ψ.Realize a → ∃ ψ ∈ Sigma, ψ.Realize a) :
    ∃ (β : Finset L[[α]].Sentence) (f : β → Sigma),
    T ⊨ᵇ ψ.imp (iSup (Subtype.val ∘ f))
```

3.2 Formal proof of the Separating Types Theorem

Now, we can prove the Separating Types Theorem. In natural language, the proof looks as follows:

Proof of theorem 3.1.
(1) ⇒ (2): Let p, q be distinct n-types over T. Then, there is an $\mathcal{L}$-formula $\varphi(x_1, \ldots, x_n)$ such that $\varphi \in p$ and $\varphi \notin q$. Moreover, by (1), there is an $\mathcal{L}$-formula $\psi(x_1, \ldots, x_n) \in \Sigma$ such that $T \vDash (\varphi \leftrightarrow \psi)$. Hence, $\psi \in p$ and $\psi \notin q$, which is what we wanted to show.

(2) ⇒ (1): Let $\varphi(x_1, \ldots, x_n)$ be an $\mathcal{L}$-formula.

Claim 3.5. For every complete type $p \ni \varphi$ over T, there exists an $\mathcal{L}$-formula $\chi_p \in p \cap \Sigma$ such that $T \vDash (\chi_p \to \varphi)$.

Proof of Claim 3.5. Let $\Sigma' := \{\neg\psi : \psi \in \Sigma \cap p\}$. Moreover, let $\mathcal{M} \vDash T$ and $a \in M^n$ such that $\mathcal{M} \models \neg\varphi(a)$.
This means that $\neg\varphi \in \mathrm{tp}^{\mathcal{M}}(a)$, which implies $\mathrm{tp}^{\mathcal{M}}(a) \neq p$. Using (2), we get that there exists $\psi \in \Sigma$ such that $\psi \in p$ and $\psi \notin \mathrm{tp}^{\mathcal{M}}(a)$. Therefore, $\neg\psi \in \mathrm{tp}^{\mathcal{M}}(a)$, meaning that $\mathcal{M} \models \neg\psi(a)$.

Hence, the assumptions of Lemma 3.3 are satisfied, meaning that there are finitely many $\mathcal{L}$-formulas $\psi_1, \ldots, \psi_m \in \Sigma \cap p$ such that

$$T \vDash (\neg\varphi) \to (\neg\psi_1 \vee \neg\psi_2 \vee \ldots \neg\psi_m). \tag{3.1}$$

Now, let $\chi_p := \bigwedge_{i=1}^{m} \psi_i$. Since Σ is closed under conjunctions, we know that $\chi_p \in \Sigma \cap p$. Moreover, by (3.1), we know that $T \vDash \neg\varphi \to \neg\chi_p$, meaning that $T \vDash \chi_p \to \varphi$, which is what we wanted to show. □

Now, define

$$\Delta := \{\chi_p : p \text{ complete } n\text{-type over } T, \varphi \in p\} \tag{3.2}$$

to be the set of all χ_p that are defined as above.
Now, let $\mathcal{M} \models T$ and $a \in M^n$ be such that $\mathcal{M} \models \varphi(a)$. Then, we know that $\varphi \in \mathrm{tp}^{\mathcal{M}}(a)$. Since $T \vDash (\varphi \to \chi_{\mathrm{tp}^{\mathcal{M}}(a)})$, it follows that $\mathcal{M} \vDash \chi_{\mathrm{tp}^{\mathcal{M}}(a)}(a)$.

Hence, we can apply Lemma 3.3, meaning that there are $\xi_1, \ldots, \xi_l \in \Delta$ such that

$$T \vDash \left(\varphi \to \bigvee_{i=1}^{l} \xi_i\right). \tag{3.3}$$

Note that by definition of Δ, we have $\xi_i \in \Sigma$ for all $1 \leq i \leq l$, meaning that $\bigvee_{i=1}^{l} \xi_i \in \Sigma$. Moreover, for all $1 \leq i \leq l$, $T \vDash (\xi_i \to \varphi)$ holds by definition of Δ. Hence, we get

$$T \vDash \left(\varphi \leftrightarrow \bigvee_{i=1}^{m} \xi_i\right), \tag{3.4}$$

which is what we wanted to prove. □

In order to formalize the proof above, we first formalized some basic properties of complete types and sets of formulas with certain properties, in particular:

- two complete types p, q over T are distinct if and only if there is an $\mathcal{L}$-formula that is contained in p, but not in q (`types_neq_iff` in the author's formalization project)
- A set Σ of formulas that is closed under conjunction and contains $\top$ also contains the conjunction of finitely many elements of Σ (`closed_inf_contains_iInf` in the formalization project). Similarly, a set Σ of formulas that is closed under disjunction and contains $\bot$ also contains the disjunction of finitely many elements of Σ (`closed_sup_contains_iSup`)
- Complete types over T are closed under conjunction and disjunction (`CompleteType.closed_inf` and `CompleteType.closed_sup` in the formalization project)

- If p is a complete type over T and φ is an $\mathcal{L}$-formula with $T \vDash \varphi$, φ is contained in p (`CompleteType.contains_modeled_formulas` in the formalization project) and
- If $T \vDash (\varphi \leftrightarrow \psi)$, then a complete type p over T contains φ if and only if it contains ψ (`CompleteType.contains_equiv` in the formalization project).

Again, we omit the formal proofs here.

Now, we are ready to formalize Claim 3.5. Again, due to the length of the formal proof, we only refer to the Lean code on GitHub instead of presenting the proof on paper.

```
-- Code in the file SeparatingTypes of the author's formalization project
lemma contains_formula_implies {α : Type w} [Finite α] (T : L.Theory)
  (Sigma : Set (L.Formula α))
  (closed_sup : ∀ φ ψ, (φ ∈ Sigma) ∧ (ψ ∈ Sigma) → (φ ⊔ ψ) ∈ Sigma)
  (closed_inf : ∀ φ ψ, (φ ∈ Sigma) ∧ (ψ ∈ Sigma) → (φ ⊓ ψ) ∈ Sigma)
  (contains_top : ⊤ ∈ Sigma) (separation : ∀ p q : T.CompleteType α,
  p.toTheory ≠ q.toTheory → ∃ ψ : L.Formula α,
  ψ ∈ Sigma ∧ equivSentence ψ ∈ p.toTheory ∧
  ¬equivSentence ψ ∈ q.toTheory) :
    ∀ φ : L.Formula α, ∀ p : T.CompleteType α,
    (equivSentence φ) ∈ p.toTheory → ∃ χ_p ∈ Sigma,
    (equivSentence χ_p) ∈ p.toTheory ∧ T ⊨ᵇ χ_p.imp φ
```

Using this lemma, we formalized the Separating Types Theorem. For the sake of readability, we skipped some details of the formalization here. For the full proof, see the author's project on GitHub.

```
-- Code in the file SeparatingTypes
-- of the author's formalization project
theorem ContainsEquivForumulas_iff_SeparatesTypes {α : Type w} [Finite α]
  (T : L.Theory) (Sigma : Set (L.Formula α))
  (closed_sup : ∀ φ ψ, (φ ∈ Sigma) ∧ (ψ ∈ Sigma) → (φ ⊔ ψ) ∈ Sigma)
  (closed_inf : ∀ φ ψ, (φ ∈ Sigma) ∧ (ψ ∈ Sigma) → (φ ⊓ ψ) ∈ Sigma)
  (contains_top : ⊤ ∈ Sigma) (contains_bot : ⊥ ∈ Sigma) :
    (∀ φ : L.Formula α, ∃ ψ ∈ Sigma, T.Iff φ ψ) ↔
    (∀ p q : T.CompleteType α, p.toTheory ≠ q.toTheory →
    ∃ ψ : L.Formula α, ψ ∈ Sigma ∧ equivSentence ψ ∈ p.toTheory ∧
    ¬equivSentence ψ ∈ q.toTheory) := by
      constructor
      · -- Suppose for every L-formula φ with variables in α,
        -- there is an L-formula ψ ∈ Sigma that is equivalent
        -- to φ modulo T. Then, let p and q be
        -- distinct complete types.
        intro hSigma p q hpq
        -- Then, there is an L-formula φ s.t. φ ∈ p and φ ∉ q.
        apply (types_neq_iff T p q).1 at hpq
        obtain ⟨ φ, hφ₁, hφ₂ ⟩ := hpq
        let φ' := equivSentence.invFun φ
```

```
  -- By assumption, there is ψ ∈ Sigma
  -- that is equivalent to φ (modulo T).
  specialize hSigma φ'
  obtain ⟨ ψ, hψ ⟩ := hSigma
  -- We are going to show that ψ ∈ p and ψ ∉ q.
  use ψ
  have hφ' : equivSentence φ' = φ :=
    (Equiv.apply_eq_iff_eq_symm_apply
    equivSentence).mpr rfl
  rw [← hφ'] at hφ₁; rw [← hφ'] at hφ₂
  apply (contains_equiv T p φ' ψ hψ.2).1 at hφ₁
  constructor
  · exact hψ.1
  · constructor
    · -- Since ψ is equivalent to φ,
      -- it is clear that ψ ∈ p.
      gcongr
    · -- By the same argument, we get that ψ ∉ q.
      by_contra h'
      apply hφ₂
      apply (contains_equiv T q φ' ψ hψ.2).2
      gcongr
· /- Now, suppose that for all distinct
  complete types p and q,
  there is ψ ∈ Sigma s.t. ψ ∈ p and ψ ∉ q.
  Let φ be an L-formula with variables in α.
  We are going to show that there is
  ψ ∈ Sigma s.t. φ and ψ
  are equivalent modulo T. -/
  intro hSigma φ
  /- By claim 3.5, for all complete types p,
  there is χ_p ∈ p ∩ Sigma s.t. T ⊨ (χ_p → φ).-/
  have hSigma₁ : ∀ p : T.CompleteType α,
    (equivSentence φ) ∈ p.toTheory →
    ∃ χ_p ∈ Sigma, (equivSentence χ_p) ∈ p.toTheory
    ∧ T ⊨ᵇ χ_p.imp φ :=
      contains_formula_implies
        T Sigma closed_sup closed_inf
        contains_top hSigma φ
  /- We define Delta to be the set of all such χ_ps
  (More precisely, Delta contains
  exactly one formula χ_p for every type p containing φ). -/
  let Delta : Set (L.Formula α) :=
    {χ_p | ∃ p : T.CompleteType α,
      ∃ (hp :(equivSentence φ) ∈ p.toTheory),
      χ_p = Classical.choose (hSigma₁ p hp)}
  /- Now, let us prove that for every M ⊨ T
  and every a : α → M with M ⊨ φ(a),
  there is a ψ ∈ Delta with M ⊨ ψ(a): -/
  have hDelta : ∀ (M : ModelType.{u, v, max (max u v) w} T)
    (a : α → M.Carrier), φ.Realize a →
    ∃ ψ ∈ Delta, ψ.Realize a := sorry
  /- Applying Lemma 3.3 yields that there is a
  finite subset of Delta s.t. ¬φ implies
```

```
the disjunction of the elements of that subset. -/
have hDelta'' :=
  impliesfinitedisj_if T ψ Delta hDelta
obtain ⟨ β, f, hDelta'' ⟩ := hDelta''
/- Let ψ be the disjunction of the elements of
the finite subset of Delta -/
let ψ := Formula.iSup (Subtype.val ∘ f)
-- By definition of Delta, we know that Delta ⊆ Sigma.
have hDelta₁ : Delta ⊆ Sigma := sorry
/- Hence, ψ ∈ Sigma since ψ is a disjunction of finitely
many elements of Sigma and Sigma is
closed under disjunction. -/
have hψ₁ : ψ ∈ Sigma := sorry
-- Now, let us prove that T ⊨ (ψ → ψ):
have hψ₂ : T ⊨[b] ψ.imp ψ := sorry
-- Hence, we know that T ⊨ (ψ → ψ) and T ⊨ (ψ → ψ),
-- therefore T ⊨ (ψ → ψ), which is what we wanted to prove.
use ψ
constructor
· exact hψ₁
· unfold Theory.Iff
  tauto
```

4 Back-and-Forth-Systems and Quantifier Elimination

The goal of this section is to formalize Theorem 1.2.
In order to do this, we need to define quantifier elimination, partial isomorphisms and back-and-forth-systems in Lean.

4.1 Partial Isomorphisms and Back-and-Forth-Systems

Definition 4.1. Let $\mathcal{M}$ and $\mathcal{N}$ be $\mathcal{L}$-structures. A bijection $\iota : A \to B$ with $A \subseteq M$, $B \subseteq N$ is called a *partial isomorphism* if

1. For every n-ary function symbol $f \in \mathcal{L}$ and $a_1, \ldots, a_{n+1} \in A$ with $f^{\mathcal{M}}(a_1, \ldots, a_n) = a_{n+1}$, we have
$$f^{\mathcal{N}}(\iota(a_1), \ldots, \iota(a_n)) = \iota(a_{n+1}).$$
2. For every n-ary function symbol $R \in \mathcal{L}$ and $a_1, \ldots, a_n \in A$,
$$R^{\mathcal{M}}(a_1, \ldots, a_n) \text{ holds if and only if } R^{\mathcal{N}}(\iota(a_1), \ldots, \iota(a_n))$$
.

Here, $f^{\mathcal{M}}$ (resp. $R^{\mathcal{M}}$) is defined to be the interpretation of f (resp. R) in $\mathcal{M}$. $f^{\mathcal{N}}$ and $R^{\mathcal{N}}$ are defined analogously.

In Lean, we formalized this definition as follows:

```
-- Code in the file QuantifierElimination
-- of the author's formalization project
structure PartialIso extends (_root_.PartialEquiv M N) where
  /-- A partial isomorphism commutes with interpretation of functions -/
  map_fun' : ∀ {n}, ∀ (f : L.Functions n), ∀ (a : Fin n → M), ∀ (m : M),
    ((∀ x, a x ∈ source) ∧ m ∈ source) →
    (Structure.funMap f a = m ↔ Structure.funMap f (toFun ∘ a) = toFun m)
  /-- A partial isomorphism commutes with interpretation of relations -/
  map_rel' : ∀ {n}, ∀ (r : L.Relations n),
    ∀ (a : Fin n → M), (∀ x, a x ∈ source) →
    (Structure.RelMap r a ↔ Structure.RelMap r (toFun ∘ a))
```

Note: `_root_.PartialEquiv` is defined as follows:

```
-- Code in the mathlib file Logic/Equiv/PartialEquiv,
-- written by Sébastien Gouëzel
structure PartialEquiv (α : Type*) (β : Type*) where
  /-- The global function which has a partial inverse.
  Its value outside of the `source` subset is irrelevant. -/
  toFun : α → β
  /-- The partial inverse to `toFun`. Its value outside
  of the `target` subset is irrelevant. -/
  invFun : β → α
  /-- The domain of the partial equivalence. -/
  source : Set α
  /-- The codomain of the partial equivalence. -/
  target : Set β
  /-- The proposition that elements of `source`
  are mapped to elements of `target`. -/
  map_source' : ∀ ⦃x⦄, x ∈ source → toFun x ∈ target
  /-- The proposition that elements of `target`
  are mapped to elements of `source`. -/
  map_target' : ∀ ⦃x⦄, x ∈ target → invFun x ∈ source
  /-- The proposition that `invFun` is a
  left-inverse of `toFun` on `source`. -/
  left_inv' : ∀ ⦃x⦄, x ∈ source → invFun (toFun x) = x
  /-- The proposition that `invFun` is a
  right-inverse of `toFun` on `target`. -/
  right_inv' : ∀ ⦃x⦄, x ∈ target → toFun (invFun x) = x
```

Essentially, in Lean, `PartialIso M N` is not defined as a function from a subset of M to a subset of N, but as a function `toFun : M → N` together with an "inverse function" `invFun : N → M`, but `toFun` is only required to be bijective if restricted to a function from A to B. On B, `invFun` is indeed the inverse of `toFun`, but the values of `toFun` and `invFun` outside of A resp. B are irrelevant.

Remark 4.2. `PartialIso M N` is implemented this way to avoid coercions between the types M and A resp. N and B. Generally, when using Lean, it is encouraged to avoid the usage of subtypes whenever possible.

In order to prove our main result, we need the notion of a back-and-forth system:

Definition 4.3. Let $\mathcal{M}, \mathcal{N}$ be $\mathcal{L}$-structures. A *back-and-forth-system* (short: *BNF-system*) $\mathcal{F}$ between $\mathcal{M}$ and $\mathcal{N}$ is a set of partial isomorphisms between $\mathcal{M}$ and $\mathcal{N}$ such that

(back) For every $\iota : A \to B \in \mathcal{F}$ and every $b \in N$, there exists $\iota' : A' \to B' \in \mathcal{F}$ such that $A \subseteq A'$, $B \subseteq B'$ and $\iota'|_A = \iota$ and $b \in B'$.

(forth) For every $\iota : A \to B \in \mathcal{F}$ and every $a \in M$, there exists $\iota' : A' \to B' \in \mathcal{F}$ such that $A \subseteq A'$, $B \subseteq B'$ and $a \in A'$.

In Lean, this is formalized as follows:

```
-- Code in the file QuantifierElimination of
-- the author's formalization project
def IsBackAndForthSystem (F : Set (L.PartialIso M N)) : Prop :=
  (∀ f ∈ F, ∀ (m : M), ∃ g ∈ F, m ∈ g.source ∧ f ≤ g) ∧
  (∀ f ∈ F, ∀ (n : N), ∃ g ∈ F, n ∈ g.target ∧ f ≤ g)
```

Here, `f ≤ g` holds for two partial isomorphisms `f` and `g` if the domain of `f` is contained in the domain of `g` and `f` and `g` are equal on the domain of `f`:

```
-- Code in the file QuantifierElimination of
-- the author's formalization project
instance : LE (L.PartialIso M N) :=
 ⟨ fun f g ↦
 (f.source ⊆ g.source ∧ (∀ x ∈ f.source, f.toFun x = g.toFun x))⟩
```

Remark 4.4. Our implementation of the "forth"-direction of back-and-forth-system is based on Gabin Kolly's definition of *extension pairs*, which he used to make a back-and-forth-argument similar to Theorem 1.2 in his formalization work on Fraissë-limits [Kol25]:

```
-- Code in the mathlib file ModelTheory/PartialEquiv,
-- written by Gabin Kolly
def IsExtensionPair : Prop :=
  ∀ (f : L.FGEquiv M N) (m : M), ∃ g, m ∈ g.1.dom ∧ f ≤ g
```

Here, two $\mathcal{L}$-structures $\mathcal{M}$ and $\mathcal{N}$ are said to form an extension pair if every equivalence between finitely generated substructures of $\mathcal{M}$ and $\mathcal{N}$ can be extended similar to the "forth"-direction of the previous definition.
While we will indeed consider the set of partial isomorphisms with finite domain later, one should note that our definition of partial isomorphisms is more general than `PartialEquiv`, which is implicitly used in `IsExtensionPair`, since `PartialEquiv` requires the domain and codomain to be substructures of $\mathcal{M}$ respectively $\mathcal{N}$, while `PartialIso` only requires them to be subsets.

One can easily see from the definition of a back-and-forth-system that one can extend the domain (resp. codomain) of a partial isomorphism ι in a BNF-system to include any finite number of elements of M (resp. N) instead of just one. In Lean, this is formalized as

```
-- Code in the file QuantifierElimination of
-- the author's formalization project
lemma forth_extends_finite (F : Set (L.PartialIso M N)) (n : ℕ) :
  (L.IsBackAndForthSystem M N F) →
  ∀ f ∈ F, ∀ (v : Fin n → M), ∃ g ∈ F,
  (∀ (a : Fin n), v a ∈ g.source) ∧ f ≤ g := sorry

lemma back_extends_finite (F : Set (L.PartialIso M N)) (n : ℕ) :
  (L.IsBackAndForthSystem M N F) →
  (∀ f ∈ F, ∀ (v : Fin n → N), ∃ g ∈ F,
  (∀ (a : Fin n), v a ∈ g.target) ∧ f ≤ g) := sorry
```

The straightforward induction proofs are omitted here.

A key result about back-and-forth systems is the following one:

Theorem 4.5. *Let $\mathcal{M}$ and $\mathcal{N}$ be two $\mathcal{L}$-structures. Suppose there exists a nonempty back-and-forth-system $\mathcal{F}$ between $\mathcal{M}$ and $\mathcal{N}$. Then, $\mathcal{M}$ and $\mathcal{N}$ are elementary equivalent.*

To prove this theorem, we are going to prove that application of partial isomorphisms within a back-and-forth-system is compatible with interpretation of terms and formulas:

Lemma 4.6. *Let $\mathcal{M}$ and $\mathcal{N}$ be $\mathcal{L}$-structures and $\mathcal{F}$ be a back-and-forth-system between $\mathcal{M}$ and $\mathcal{N}$. Moreover, let $t(x_1, \ldots, x_n)$ be an $\mathcal{L}$-term. Then, for all $\iota : A \to B \in \mathcal{F}$ and all $a_1, \ldots, a_{n+1} \in A$, we have*

$$t^{\mathcal{M}}(a_1, \ldots, a_n) = a_{n+1} \text{ if and only if } t^{\mathcal{N}}(\iota(a_1), \ldots, \iota(a_n)) = \iota(a_{n+1})$$

Proof. We show the statement by induction over terms.
If t is a variable symbol, the statement is clear.
Hence, suppose that there are a function symbol f of arity m and $\mathcal{L}$-terms $t_1(x_1, \ldots, x_n)$, ..., $t_m(x_1, \ldots, x_n)$ such that

$$t(x_1, \ldots, x_n) = f(t_1(x_1, \ldots, x_n), \ldots, t_m(x_1, \ldots, x_n))$$

holds and the statement is true for all t_i with $1 \leq i \leq m$.
Now, suppose first that $t^{\mathcal{M}}(a_1, \ldots, a_n) = a_{n+1}$. Then, since $\mathcal{F}$ is a back-and-forth-system, there is a $\iota' : A' \to B' \in \mathcal{F}$ extending ι s.t.

$$t_1(a_1, \ldots, a_n), \ldots, t_m(a_1, \ldots, a_n) \in A'.$$

Then, by the induction hypothesis and by definition of partial isomorphisms and BNF-systems, we get

$$\begin{aligned}
t^{\mathcal{N}}(\iota(a_1), \ldots, \iota(a_n)) &= t^{\mathcal{N}}(\iota'(a_1), \ldots, \iota'(a_n)) \\
&= f^{\mathcal{N}}(t_1^{\mathcal{N}}(\iota'(a_1), \ldots, \iota'(a_n)), \ldots, t_m^{\mathcal{N}}(\iota'(a_1), \ldots, \iota'(a_n))) \\
&= f^{\mathcal{N}}(\iota'(t_1^{\mathcal{M}}(a_1, \ldots, a_n)), \ldots, \iota'(t_m^{\mathcal{M}}(a_1, \ldots, a_n))) \\
&= \iota'(f^{\mathcal{M}}(t_1^{\mathcal{M}}(a_1, \ldots, a_n), \ldots, t_m^{\mathcal{M}}(a_1, \ldots, a_n))) \\
&= \iota'(t^{\mathcal{M}}(a_1, \ldots, a_n)) \\
&= \iota'(a_{n+1}) \\
&= \iota(a_{n+1}),
\end{aligned}$$

which is what we wanted to show.
On the other hand, suppose that $t^{\mathcal{N}}(\iota(a_1), \ldots, \iota(a_n)) = \iota(a_{n+1})$. Then, we can extend ι in the same way as above. By the same argument as above, we can deduce

$$\iota'(t^{\mathcal{M}}(a_1, \ldots, a_n)) = \iota'(a_{n+1}).$$

Since ι' is a bijection, this implies $t^{\mathcal{M}}(a_1, \ldots, a_n) = a_{n+1}$. □

In Lean, we formalized this lemma as follows:

```
-- Code in the file QuantifierElimination of
-- the author's formalization project
lemma on_term (F : Set (L.PartialIso M N))
  (hF : L.IsBackAndForthSystem M N F) (t : L.Term α) :
    ∀ (v : α → M) (ι : L.PartialIso M N) (hι : ι ∈ F)
    (hv₁ : ∀ x, v x ∈ ι.source) (hv₂ : t.realize v ∈ ι.source)
    (m : M) (hm : m ∈ ι.source),
    t.realize v = m ↔ t.realize (ι.toFun ∘ v) = ι.toFun m
```

The proof is ommited here, but closely mirrors the natural-language-proof above. Again, the details can be found on the author's GitHub.

Now, we can prove the compatibility with interpretation of formulas:

Lemma 4.7. *Let $\mathcal{M}$ and $\mathcal{N}$ be $\mathcal{L}$-structures and $\varphi(x_1, \dots, x_n)$ be an $\mathcal{L}$-formula. Then, for all $\iota : A \to B \in \mathcal{F}$ and all $a_1, \dots, a_{n+1} \in A$, we have*

$$\mathcal{M} \vDash \varphi(a_1, \dots, a_n) \text{ if and only if } \mathcal{N} \vDash \varphi(\iota(a_1), \dots, \iota(a_n)).$$

Proof. We show the statement by induction over formulas, using an inductive definition of formulas that is analogous to the definition in mathlib.

($\bot$) Suppose φ is $\bot$. Then, for any $a_1, \dots, a_n \in A$, $\mathcal{M} \nvDash \varphi(a_1, \dots, a_n)$ and $\mathcal{N} \nvDash \varphi(\iota(a_1), \dots, \iota(a_n))$. This is what we wanted to show.

($t_1 = t_2$) Suppose φ is $t_1(x_1, \dots, x_n) = t_2(x_1, \dots, x_n)$ for $\mathcal{L}$-terms t_1, t_2. Then, for all $a_1, \dots, a_n \in A$,

$$\mathcal{M} \vDash \varphi(a_1, \dots, a_n) \Leftrightarrow t_1^{\mathcal{M}}(a_1, \dots, a_n) = t_2^{\mathcal{M}}(a_1, \dots, a_n)$$

Moreover, since $\mathcal{F}$ is a back-and-forth-system, there is a $\iota' \in \mathcal{F}$ extending ι s.t. $t_1^{\mathcal{M}}(a_1, \dots, a_n)$ and $t_2^{\mathcal{M}}(a_1, \dots, a_n)$ are in the domain of ι'.
Since ι' is a bijection, we get

$$\begin{aligned}
& t_1^{\mathcal{M}}(a_1, \dots, a_n) = t_2^{\mathcal{M}}(a_1, \dots, a_n) \\
\Leftrightarrow\ & \iota'(t_1^{\mathcal{M}}(a_1, \dots, a_n)) = \iota'(t_1^{\mathcal{M}}(a_1, \dots, a_n)) \\
\overset{\text{Lemma 4.6}}{\Leftrightarrow}\ & t_1^{\mathcal{N}}(\iota'(a_1), \dots, \iota'(a_n)) = t_1^{\mathcal{N}}(\iota'(a_1), \dots, \iota'(a_n)) \\
\overset{\iota' \text{ extends } \iota}{\Leftrightarrow}\ & t_1^{\mathcal{N}}(\iota(a_1), \dots, \iota(a_n)) = t_1^{\mathcal{N}}(\iota(a_1), \dots, \iota(a_n)) \\
\Leftrightarrow\ & \mathcal{N} \vDash \varphi(\iota(a_1), \dots, \iota(a_n)),
\end{aligned}$$

which is what we wanted to prove.

$(\varphi_1 \to \varphi_2)$ Suppose φ is an implication of two $\mathcal{L}$-formulas $\varphi_1(x_1, \ldots, x_n)$ and $\varphi_2(x_1, \ldots, x_n)$ for which the statement holds.
Then, for all $a_1, \ldots, a_n \in A$,

$$\begin{aligned}
&\mathcal{M} \vDash \varphi(a_1, \ldots, a_n) \\
\Leftrightarrow &(\mathcal{M} \vDash \varphi_1(a_1, \ldots, a_n) \Rightarrow \mathcal{M} \vDash \varphi_2(a_1, \ldots, a_n)) \\
\overset{\text{Induction hypothesis}}{\Leftrightarrow} &(\mathcal{N} \vDash \varphi_1(\iota(a_1), \ldots, \iota(a_n) \Rightarrow \mathcal{N} \vDash \varphi_2(\iota(a_1), \ldots, \iota(a_n)))) \\
\Leftrightarrow &\mathcal{N} \vDash \varphi(\iota(a_1), \ldots, \iota(a_n)),
\end{aligned}$$

which is what we wanted to show.

$(R(t_1, \ldots, t_m))$ Suppose φ is of the form $R(t_1(x_1, \ldots, x_n), \ldots, t_m(x_1, \ldots, x_n))$. Then, fix $a_1, \ldots, a_n$.
Since $\mathcal{F}$ is a back-and-forth-system, there is a $\iota' \in \mathcal{F}$ extending ι such that $t_i^{\mathcal{M}}(a_1, \ldots, a_n)$ is in the domain of ι' for all $1 \leq i \leq m$.
Since ι' is compatible with the interpretation of relation symbols, we get

$$\begin{aligned}
&\mathcal{M} \vDash \varphi(a_1, \ldots, a_n) \\
\Leftrightarrow &R^{\mathcal{M}}(t_1^{\mathcal{M}}(a_1, \ldots, a_n), \ldots, t_m^{\mathcal{M}}(a_1, \ldots, a_n)) \\
\overset{\iota' \text{ is a partial iso}}{\Leftrightarrow} &R^{\mathcal{N}}(\iota'(t_1^{\mathcal{M}}(a_1, \ldots, a_m)), \ldots, \iota'(t_m^{\mathcal{M}}(a_1, \ldots, a_n))) \\
\overset{\text{Lemma 4.6}}{\Leftrightarrow} &R^{\mathcal{N}}(t_1^{\mathcal{N}}(\iota'(a_1), \ldots, \iota'(a_m)), \ldots, t_m^{\mathcal{N}}(\iota'(a_1), \ldots, \iota'(a_n))) \\
\overset{\iota' \text{ extends } \iota}{\Leftrightarrow} &R^{\mathcal{N}}(t_1^{\mathcal{N}}(\iota(a_1), \ldots, \iota(a_m)), \ldots, t_m^{\mathcal{N}}(\iota(a_1), \ldots, \iota(a_n))) \\
\Leftrightarrow &\mathcal{N} \vDash \varphi(\iota(a_1), \ldots, \iota(a_n))
\end{aligned}$$

$(\forall x \varphi')$ Suppose φ is of the form $\forall x_{n+1}, \varphi'(x_1, \ldots, x_n, x_{n+1})$ for some $\mathcal{L}$-formula φ' for which the statement holds.
$\underline{\Rightarrow:}$ Suppose $\mathcal{M} \vDash \varphi(a_1, \ldots, a_n)$. We need to prove that for all $b \in N$, $\mathcal{N} \vDash \varphi(\iota(a_1), \ldots, \iota(a_n), b)$ holds.
Fix $b \in N$. Then, since $\mathcal{F}$ is a back-and-forth-system, there exists $\iota' \in \mathcal{F}$ extending ι s.t. b is in the codomain of ι'. Let a be the preimage of b under ι'.
The statement holding for φ' yields

$$\begin{aligned}
&\mathcal{M} \vDash \varphi(a_1, \ldots, a_n) \\
\Rightarrow &\mathcal{M} \vDash \varphi'(a_1, \ldots, a_n, a) \\
\Leftrightarrow &\mathcal{N} \vDash \varphi'(\iota'(a_1), \ldots, \iota'(a_n), \iota'(a)) \\
\Leftrightarrow &\mathcal{N} \vDash \varphi'(\iota'(a_1), \ldots, \iota'(a_n), b) \\
\overset{\iota' \text{ extends } \iota}{\Leftrightarrow} &\mathcal{N} \vDash \varphi'(\iota(a_1), \ldots, \iota(a_n), b),
\end{aligned}$$

which proves the forward direction.
$\underline{\Leftarrow:}$ Suppose $\mathcal{N} \vDash \varphi(\iota(a_1), \ldots, \iota(a_n))$ We need to prove that for all

$a \in M$, $\mathcal{M} \vDash \varphi'(a_1, \dots, a_n, a)$.
Fix $a \in M$. Then, since $\mathcal{F}$ is a back-and-forth-system, there exists $\iota' \in \mathcal{F}$ extending ι such that a is in the domain of ι'. Let $b := \iota'(a)$.
The statement holding for φ' yields

$$\begin{aligned}
&\mathcal{N} \vDash \varphi(\iota(a_1), \dots, \iota(a_n)) \\
\Rightarrow &\mathcal{N} \vDash \varphi'(\iota(a_1), \dots, \iota(a_n), b) \\
\Leftrightarrow &\mathcal{N} \vDash \varphi'(\iota(a_1), \dots, \iota(a_n), \iota'(a)) \\
\overset{\iota' \text{ extends } \iota}{\Leftrightarrow} &\mathcal{N} \vDash \varphi'(\iota'(a_1), \dots, \iota'(a_n), \iota'(a)) \\
\overset{\text{induction hypothesis}}{\Leftrightarrow} &\mathcal{M} \vDash \varphi(a_1, \dots, a_n, a),
\end{aligned}$$

which proves the backwards direction.

□

In Lean, we need to prove the statement first for bounded formulas and then for formulas:

```
-- Code in the file QuantifierElimination
-- of the author's formalization project
theorem on_boundedFormula {n : ℕ} (F : Set (L.PartialIso M N))
  (hF : L.IsBackAndForthSystem M N F) (φ : L.BoundedFormula α n) :
    ∀ (v : α → M) (xs : Fin n → M) (ι : L.PartialIso M N)
    (hι : ι ∈ F) (hv : ∀ x, v x ∈ ι.source) (hxs : ∀ x, xs x ∈ ι.source),
    φ.Realize v xs ↔ φ.Realize (ι.toFun ∘ v) (ι.toFun ∘ xs)

theorem on_formula (F : Set (L.PartialIso M N))
  (hF : L.IsBackAndForthSystem M N F) (φ : L.Formula α) :
    ∀ (v : α → M) (ι : L.PartialIso M N) (hι : ι ∈ F)
    (hv : ∀ x, v x ∈ ι.source),
φ.Realize v ↔ φ.Realize (ι.toFun ∘ v)
```

The proofs are omitted here, but the formal proof of the first statement is completely analogous to the natural-language-proof above, while the second statement immediately follows from the first one by using the definition of formulas in Lean.

Now, Theorem 4.5 follows as a corollary:

Proof of Theorem 4.5. Let $\mathcal{M}, \mathcal{N}$ be $\mathcal{L}$-structures and $\mathcal{F}$ be a nonempty back-and-forth-system between $\mathcal{M}$ and $\mathcal{N}$.
We need to prove that for every $\mathcal{L}$-sentence φ, $\mathcal{M} \vDash \varphi$ if and only if $\mathcal{N} \vDash \varphi$.
Now, let $\iota : A \to B$ be a partial isomorphism in $\mathcal{F}$. Then, φ (considered as a formula) is realized by an n-tuple in A if and only if it is realized by the images of the elements of this n-tuple under ι.
Since φ is a sentence, i.e. a formula without free variables, $n = 0$, i.e. the previous statement translates to $\mathcal{M} \vDash \varphi \Leftrightarrow \mathcal{N} \vDash \varphi$. □

In Lean, this reads as follows:

```
theorem elementarilyEquivalent_if_exists_nonempty
  {L : Language} (M : Type w) (N : Type w')
  [L.Structure M] [L.Structure N]:
    (∃ (F : Set (L.PartialIso M N)), (Nonempty F ∧
    (L.IsBackAndForthSystem M N F))) →
    L.ElementarilyEquivalent M N := by
      -- Suppose `F` is a nonempty
      -- back-and-forth-system between `M` and `N`
      intro h
      obtain ⟨ F, ι, hF ⟩ := h
      apply Classical.choice at ι
      -- We need to show that for every `L`-sentence `ψ`,
      -- `M ⊨ ψ` if and only if `N ⊨ ψ`.
      apply elementarilyEquivalent_iff.2
      intro ψ
      unfold Sentence at ψ
      let v : Empty → M := default
      -- Since `F` is nonempty,
      -- there exists a partial iso `ι' ∈ F`.
      let ι' : L.PartialIso M N := ι
      have hι' : ι' ∈ F := Subtype.coe_prop ι
      have hv : ∀ (x : Empty), v x ∈ ι'.source := by simp
      unfold Sentence.Realize
      have hv₁ : v = default := Unique.ext_iff.mp rfl
      have hv₂ : ι'.toFun ∘ v = default := by
        ext x; trivial
      rw [← hv₁, ← hv₂]
      -- The goal follows from the fact that `ι'` is compatible
      -- with interpretation of formulas.
      exact on_formula M N F hF ψ v ι' hι' hv
```

4.2 Definition of Quantifier Elimination

As already noted in the introduction, quantifier elimination is defined as follows:

Definition 4.8. Let $\mathcal{L}$ be a language. An $\mathcal{L}$-theory T is said to have *quantifier elimination* if for every $\mathcal{L}$-formula $\varphi(x_1, \ldots, x_n)$, there exists a quantifier-free $\mathcal{L}$-formula $\psi(x_1, \ldots, x_n)$ such that $T \vDash (\varphi \leftrightarrow \psi)$.

In Lean, we defined QE as follows:

```
-- Code from the file QuantifierElimination
-- of the author's formalization project
def HasQE {L : Language} (T : L.Theory) : Prop :=
  ∀ (α : Type*) (_ : DecidableEq α) (φ : L.Formula α),
  ∃ ψ : L.Formula α, (ψ.IsQF ∧ T ⊨ᵇ (φ.iff ψ))
```

Note: for a number of quantifier elimination criteria, including the one we are formalizing in this thesis, we need to use that any $\mathcal{L}$-formula φ contains only finitely many free variables. But in order to obtain a `freeVarFinset`

from an $\mathcal{L}$-formula with variables in α, we need an instance of `DecidableEq α`, which is the reason why we only consider types with such a typeclass instance.

4.3 Quantifier Eliminaton test

Now, we can prove the quantifier elimination test from the introduction:

Theorem 4.9. *Let $\mathcal{L}$ be a language and T an $\mathcal{L}$-theory such that for all $\mathcal{M}, \mathcal{N} \vDash T$, the set of partial isomorphisms between $\mathcal{M}$ and $\mathcal{N}$ with finite domain is a back-and-forth-system.*
Then, T has quantifier elimination.

Proof. Let $n \in \mathbb{N}$. We are going to show that every $\mathcal{L}$-formula $\varphi(x_1, \dots, x_n)$ is equivalent to a quantifier-free $\mathcal{L}$-formula $\psi(x_1, \dots, x_n)$.

To prove this, let Σ be the set of quantifier-free $\mathcal{L}$-formulas with free variables in the set $\{x_1, \dots, x_n\}$. One can easily see that Σ contains $\top$ and $\bot$ and is closed under conjunction and disjunction of $\mathcal{L}$-formulas. Hence, by the Separating Types Theorem, it suffices to show that for any two distinct complete n-types p, q over T, there exists a quantifier-free formula $\psi(x_1, \dots, x_n)$ such that $\psi \in p$ and $\psi \notin q$.

Let p, q be complete n-types over T and suppose that for all quantifier-free formulas $\psi(x_1, \dots, x_n)$, $\psi \in p$ if and only if $\psi \in q$. Then, we are going to prove $p = q$.

Note that there exist $\mathcal{M}, \mathcal{N} \vDash T$ and $a \in M^n$, $b \in N^n$ such that a realizes p and b realizes q. Then, consider the map $\iota : \{a_1, \dots, a_n\} \to \{b_1, \dots, b_n\}$ mapping a_i to b_i for all $1 \leq i \leq n$.
Note that since a and b satisfy the same quantifier-free $\mathcal{L}$-formulas, ι is a partial isomorphism between $\mathcal{M}$ and $\mathcal{N}$. In particular, ι is an element of a back-and-forth-system.
Hence, for all $\mathcal{L}$-formulas $\varphi(x_1, \dots, x_n)$, we know by Lemma 4.7 that

$$\mathcal{M} \vDash \varphi(a) \Leftrightarrow \mathcal{N} \vDash \varphi(b).$$

This means that $\varphi \in \mathrm{tp}^{\mathcal{M}}(a) = p$ if and only if $\varphi \in \mathrm{tp}^{\mathcal{N}}(b) = q$. This concludes the proof. □

For the formal proof, we first need to construct a partial isomorphism that sends a given tuple a in an $\mathcal{L}$-structure $\mathcal{M}$ to another given tuple b in an $\mathcal{L}$-structure $\mathcal{N}$ that satisfies the same quantifier-free formulas as a.

Note that in Lean, realizations of $\mathcal{L}$-formulas with variables in α are implemented as functions `α → M`, where $\mathcal{M}$ is the $\mathcal{L}$-structure in question. Hence, for any `a : α → M` and `b : α → N` satisfying the same

quantifier-free formulas, we want to define a map `ι : M → N` that is bijective if restricted to the images of `a` and `b` and satisfies `ι (a (i)) = b (i)` for all `i ∈ α`.
Therefore, we define $\iota(m)$ as $b(a^{-1}(m))$ for all m in the image of a and as an arbitrary element of N otherwise. For this purpose, we require $\mathcal{M}$ and $\mathcal{N}$ to be nonempty.

Note that `a` is not necessarily injective, therefore, it still needs to be proven that `ι` is well-defined. But, since $x_i = x_j$ is quantifier-free formula for $i, j \in \alpha$, we know that $a(i) = a(j)$ holds if and only if $b(i) = b(j)$, therefore, the image of m under ι does not depend on the choice of the preimage under a.

Similarly, we can define the "inverse function" of ι by sending $n \in N$ to $a(b^{-1}(n))$, if n is in the preimage of b, and to an arbitrary element of m otherwise. Then, one can easily see that on the images on a resp. b, ι and its "inverse" compose to the identity.

Moreover, ι is compatible with interpretation of function and relation symbols since $f(x_1, \ldots, x_n) = x_{n+1}$ and $R(x_1, \ldots, x_n)$ are quantifier-free $\mathcal{L}$-formulas.

In Lean, the definition looks like this:

```
-- Code in the file QuantifierElimination
-- of the author's formalization project
noncomputable def CompleteType.toPartialIso {α : Type*}
  {L : Language} (M : Type w) (N : Type w')
  [L.Structure M] [L.Structure N] [Nonempty M] [Nonempty N]
  (a : α → M) (b : α → N) (hab : ∀ φ : L.Formula α, φ.IsQF →
  (φ.Realize a ↔ φ.Realize b)) :
    L.PartialIso M N where
      source := {y | ∃ x, a x = y}
      target := {y | ∃ x, b x = y}
      /- toFun maps every element of M in the image of a
      to the image under `b` of a preimage under a.
      Note: a and b are not necessarily injective,
      so we still have to prove that this map does not
      depend on the choice of the preimage under a. -/
      toFun := fun x ↦ by
        by_cases h : ∃ y, a y = x
        · use b (Classical.choose h)
        · rename_i hM₁ hN₁ hM₂ hN₂
          use (Classical.choice hN₂)
      /- invFun maps every element of N in the image of b
      to the image under a of a preimage under b
      Note: Since a and b are not necessarily injective,
      it still remains to prove that this map does not
      depend on the choice of the preimage under b. -/
      invFun := fun x ↦ by
```

```
      by_cases h : ∃ y, b y = x
      · use a (Classical.choose h)
      · rename_i hM₁ hN₁ hM₂ hN₂
        use (Classical.choice hM₂)
    map_source' := by
      intro x hx
      simp_all
    map_target' := by
      intro x hx
      simp_all
  left_inv' := sorry
  right_inv' := sorry
  map_fun' := sorry
  map_rel' := sorry
```

Note: here, when defining `toFun` and `invFun`, we do not need to show that they are well-defined. Instead, we just fix an arbitrary preimage of $m \in M$ under a (or a preimage of $n \in N$ under b, respectively) using the axiom of choice.
However, we still use the argument that was presented above when showing that `toFun` is a left-inverse of `invFun`:

```
-- Code in the file QuantifierElimination
-- of the author's formalization project
left_inv' := by
  /- Let x ∈ M. We need to prove that if x
  is in the image of a, then invFun(toFun(x)) = x. -/
  intro x hx
  /- Since equality of two variables is
  a quantifier-free formula, we know that two elements of a
  have the same image under a if and only if
  they have the same image under b. -/
  have h : ∀ i₁ i₂ : α, a i₁ = a i₂ ↔ b i₁ = b i₂ := by
    intro i₁ i₂
    let t₁ : L.Term (α ⊕ (Fin 0)) := var (Sum.inl i₁)
    let t₂ : L.Term (α ⊕ (Fin 0)) := var (Sum.inl i₂)
    let φ : L.Formula α := BoundedFormula.equal t₁ t₂
    have hφ : φ.IsQF := by
      unfold IsQF
      tauto
    specialize hab φ hφ
    finiteness
  /- Hence, for all x in the image of a,
  the image under b of any preimage under a equals x.
  This is what we wanted to prove. -/
  have h' : ∀ x, x ∈ {y | ∃ x', a x' = y} → ((fun (z : N) ↦ by
    by_cases h : ∃ y, b y = z
    · use a (Classical.choose h)
    · rename_i hM₁ hN₁ hM₂ hN₂ _ _
      use (Classical.choice hM₂)) ((fun (z : M) ↦ by
        by_cases h : ∃ y, a y = z
        · use b (Classical.choose h)
        · rename_i hM₁ hN₁ hM₂ hN₂ _ _
```

```
          use (Classical.choice hN₂)) x) = x ) := by
            simp only [Set.mem_ofPred_eq, forall_exists_index,
            forall_apply_eq_imp_iff, exists_apply_eq_apply,
            ↓reduceDIte]
            grind
  tauto
```

The proof that `invFun` is a right-inverse of `toFun` is analogous.

The proofs that `toFun` is compatible with interpretation of function and relation symbols are omitted here, but can be found on GitHub.

Note that `CompleteType.toPartialIso` is only an element of the set of partial isomorphisms with finite domain if the image of `α` under `a` is finite.
Hence, before we prove Theorem 4.9 formally, we show that if for any two models $\mathcal{M}, \mathcal{N} \vDash T$, the system of partial isomorphisms $\mathcal{M} \to \mathcal{N}$ with finite domain is a back-and-forth-system, then for every finite type `α`, every $\mathcal{L}$-formula φ with variables in `α` is equivalent modulo T to a quantifier-free formula:

```
-- Code in the file QuantifierElimination
-- of the author's formalization project
theorem HasQE_on_finite_if_BackAndForth_of_finite {L : Language}
  (T : L.Theory) :
    (∀ M N : ModelType.{u, v, max u v} T,
    L.IsBackAndForthSystem M N {f : L.PartialIso M N | Finite f.source})
    → ∀ (α : Type u) (_ : Finite α) (φ : L.Formula α),
    ∃ ψ : L.Formula α, (ψ.IsQF ∧ T ⊨ᵇ (φ.iff ψ))
```

The formal proof, the details of which can be found on GitHub, follows the natural-language-proof above closely.

In order to generalize this result to arbitrary types with decidable equality, we first proved that if `φ` is a quantifier-free $\mathcal{L}$-formulas with variables in `α`, its restriction to the set of free variables actually occuring in it is quantifier-free as well. In Lean, the statement reads:

```
theorem restrictFreeVar_IsQF {n : ℕ} {β : Type*} [DecidableEq α]
  (φ : L.BoundedFormula α n) (f : φ.freeVarFinset → β) :
  φ.IsQF ↔ (φ.restrictFreeVar f).IsQF
```

The proof is a straightforward induction over formulas.

Now, our goal follows by taking an arbitrary $\mathcal{L}$-formula φ with variables in α, considering it as a formula with free variables in a finite set (namely, the set of free variables that actually occur in φ), using the previous result to show that this is equivalent modulo T to a quantifier-free formula ψ with variables in the same finite set, and finally considering ψ as an $\mathcal{L}$-formula with variables in α.

In Lean, the statement looks as follows:

```
theorem HasQE_if_BackAndForth_of_finite {L : Language} (T : L.Theory) :
  (∀ M N : ModelType.{u, v, max u v} T,
  L.IsBackAndForthSystem M N {f : L.PartialIso M N | Finite f.source})
  → T.HasQE
```

Again, for the proof, we refer to the GitHub project.

5 Conclusion

The goal of this thesis was to formalize of the method of quantifier elimination as well as the quantifier-elimination criterion that was presented as Theorem 1.2 in the introduction in Lean, which has been done. In total, our formalization for the Separating Types Theorem and the quantifier-elimination criterion in question encompasses around 2200 lines, distributed to the two files `SeparatingTypes.lean` and `QuantifierElimination.lean` in the GitHub project.

A natural next step could be to contribute our formalization results to mathlib, which has not been done as of early August 2026. However, in order to contribute these results, the Lean code would still need to be improved, since especially the proofs of Lemma 3.3, Theorem 3.1 and Theorem 4.9 are still quite lengthy and would likely lead to a significant increase in mathlib's compilation time.

Furthermore, there are several natural ways to extend our formalization work:

First of all, one could use the formalization of Theorem 4.9 to prove quantifier elimination for several theories, like the theory of infinite sets, the theory of dense linear orders, or the theory of infinite k-vector spaces. In fact, for the case of DLO, the author has already started to formalize the quantifier-elimination result in Lean, but could not finish the formalization until the start of August 2026, which the reason that this result is not included in this thesis. For the author's current progress on formalizing the quantifier-elimination result for DLO, check the file `Applications.lean` in her project on GitHub.

Second of all, one could extend the work by formalizing other quantifier-elimination criteria, such as the standard embedding test or model companions. For the statements and proofs of these criteria, we refer to [ADH17].

Finally, as we explained in the introduction, quantifier elimination can be used to prove other model-theoretic properties of theories, most notably completeness. While mathlib already contains completeness proofs of the theory of infinite sets [And21] and the theory of dense linear orders[And22b], a completeness proof for Presburger arithmetic is still missing at the time of writing.

References

[ADH17] Matthias Aschenbrenner, Lou van den Dries, and Joris van der Hoeven. *Asymptotic Differential Algebra and Model Theory of Transseries*. Princeton: Princeton University Press, 2017, pp. 724–786. ISBN: 9781400885411. DOI: doi:10.1515/9781400885411. URL: https://doi.org/10.1515/9781400885411.

[AM20] Jeremy Avigad and Patrick Massot. *Mathematics in Lean*. 2020. URL: https://leanprover-community.github.io/mathematics_in_lean/index.html# (visited on 07/23/2026).

[And21] Aaron Anderson. *ModelTheory/Satisfiability file in the Mathlib repository*. 2021. URL: https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Satisfiability.lean (visited on 07/23/2026).

[And22a] Aaron Anderson. *ModelTheory/ElementaryMaps file in the Mathlib repository*. 2022. URL: https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/ElementaryMaps.lean (visited on 07/23/2026).

[And22b] Aaron Anderson. *ModelTheory/Order file in the Mathlib repository*. 2022. URL: https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Order.lean (visited on 07/23/2026).

[And22c] Aaron Anderson. *ModelTheory/Skolem file in the Mathlib repository*. 2022. URL: https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Skolem.lean (visited on 07/23/2026).

[And22d] Aaron Anderson. *ModelTheory/Types file in the Mathlib repository*. 2022. URL: https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/ModelTheory/Types.lean (visited on 07/23/2026).

[Avi+21] Jeremy Avigad et al. *Theorem Proving in Lean 4*. 2021. URL: https://leanprover.github.io/theorem_proving_in_lean4/ (visited on 07/21/2026).

[Bra+07] W Brauer et al. *Finite Model Theory and Its Applications*. eng. 1st ed. Texts in Theoretical Computer Science an EATCS Series. Berlin, Heidelberg: Springer Nature, 2007. ISBN: 9783540688044.

[Com] Lean Prover Community. *GitHub - leanprover/lean4: Lean 4 programming language and theorem prover*. URL: https://github.com/leanprover/lean4 (visited on 07/23/2026).

[Com17] Lean Prover Community. *GitHub - leanprover-community/mathlib4: The math library of Lean 4*. 2017. URL: https://github.com/leanprover-community/mathlib4 (visited on 07/23/2026).

[Com20a] Lean Prover Community. *Area statistics*. 2020. URL: https://leanprover-community.github.io//mathlib4_docs/mathlib.html (visited on 07/23/2026).

[Com20b] Lean Prover Community. *Mathlib statistics*. 2020. URL: https://leanprover-community.github.io/mathlib_stats.html (visited on 07/23/2026).

[Com20c] Mathlib Community. "The lean mathematical library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. DOI: 10.1145/3372885.3373824.

[Com22] Johan Commelin. "Liquid Tensor Experiment". eng. In: *Mitteilungen der Deutschen Mathematiker-Vereinigung* 30.3 (2022), pp. 166–170. ISSN: 0947-4471.

[Com23] Lean Prover Community. *Contributing to Mathlib*. 2023. URL: https://leanprover-community.github.io/contribute/.

[Cur34] H. B. Curry. "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590. DOI: 10.1073/pnas.20.11.584. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.20.11.584. URL: https://www.pnas.org/doi/abs/10.1073/pnas.20.11.584.

[Fuc10] Eugenia Fuchs. *Completeness of the random graph: two proofs*. 2010. URL: https://math.uchicago.edu/~may/VIGRE/VIGRE2010/REUPapers/Fuchs.pdf.

[HD19] Jesse Michael Han and Floris van Doorn. "A Formalization of Forcing and the Unprovability of the Continuum Hypothesis". In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 19:1–19:19. DOI: 10.4230/LIPIcs.ITP.2019.19.

[HD20] Jesse Michael Han and Floris van Doorn. "A formal proof of the independence of the continuum hypothesis". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA:

Association for Computing Machinery, 2020, pp. 353–366. DOI: 10.1145/3372885.3373826.

[Hie26] Philipp Hieronymi. "Model Theory Lecture Notes". Course notes from the lecture 'Mathematical Logic', given by Prof. Philipp Hieronymi in Winter Semester 2025/26 at the University of Bonn. Mar. 2026.

[Hod97] Wilfrid Hodges. *A shorter model theory*. USA: Cambridge University Press, 1997. ISBN: 0521587131.

[Kol25] Gabin Kolly. "Formalization of Fraïssé Limits in Lean". Available at https://www.math.uni-bonn.de/people/phierony/Kolly.pdf. Master's thesis. Bonn, Germany: Rheinische Friedrich-Wilhelms-Universität Bonn, Apr. 2025.

[Mar24] Amador Martin-Pizarro. "Non-forking independence in stable theories". eng. In: *arXiv.org* (2024). ISSN: 2331-8422.

[MU21] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5.

[NS24] Mashu Noguchi and Shogo Saito. *Formalized Formal Logic*. 2024. URL: https://github.com/FormalizedFormalLogic (visited on 07/23/2026).

[Pas16] Grzegorz Pastuszak. "A constructive proof of Tarski's theorem on quantifier elimination in the theory of ACF". eng. In: *arXiv.org* (2016). ISSN: 2331-8422.

[Ric66] J. Richard Büchi. "Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic". eng. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 44. 1966, pp. 1–11. ISBN: 0804700966.

[Sta84] Ryan Stansifer. *Presburger's Article on Integer Airthmetic: Remarks and Translation*. Tech. rep. TR84-639. Cornell University, Computer Science Department, Sept. 1984. URL: http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639.

[Szm55] Wanda Szmielew. "Elementary properties of Abelian groups". eng. In: *Fundamenta mathematicae* 41.2 (1955), pp. 228–229. ISSN: 0016-2736.

[TZ12] Katrin Tent and Martin Ziegler. *A Course in Model Theory*. Lecture Notes in Logic. Cambridge University Press, 2012. ISBN: 9781139015417.